Chapter 4—Techniques

This chapter presents Verification and Validation (V&V) techniques and provides guidelines for their use. Seventy-six V&V techniques and eighteen statistical techniques that can be used for model validation are described. Most of these techniques are derived from software engineering; the remaining are specific to the modeling and simulation field. The selected software V&V techniques applicable to Modeling and Simulation (M&S) V&V are presented in terms understandable by an M&S technical person. Some software V&V techniques are modified for use in M&S V&V. The term *testing* is used frequently in this chapter in referring to the implementation of these techniques. V&V requires the testing of the model or simulation to assess its credibility. Finally, where possible, supporting texts are referenced so that more detailed descriptions of the techniques may be obtained by the interested reader.

4.1 Verification and Validation Techniques

Figure 4-1 shows a taxonomy that lists V&V techniques in four categories: informal, static, dynamic, and formal. The use of mathematical and logical formalism in each category increases from informal to formal, from left to right. The complexity also increases as the category becomes more formal.

It should be noted that some of the categories presented in Figure 4-1 possess similar characteristics and, in fact, include techniques that overlap from one category to another. A distinct difference between each classification exists, however, as will be evident in the discussion.

4.1.1 Informal V&V Techniques

Informal techniques are among the most commonly used. They are called informal because their tools and approaches rely heavily on human reasoning and subjectivity without stringent mathematical formalism. The *informal* label does not imply, however, a lack of structure or formal guidelines in their use. In fact, these techniques are applied using well-structured approaches under formal guidelines, and they can be very effective if employed properly.

Verification and Validation Techniques

Informal
- Audit
- Desk Checking
- Face Validation
- Inspections
- Reviews
- Turing Test
- Walkthroughs

Static
- Cause-Effect Graphing
- Control Analysis
  - Calling Structure
  - Concurrent Process
  - Control Flow
  - State Transition
- Data Analysis
  - Data Dependency
  - Data Flow
- Fault/Failure Analysis
- Interface Analysis
  - Model Interface
  - User Interface
- Semantic Analysis
- Structural Analysis
- Symbolic Evaluation
- Syntax Analysis
- Traceability Assessment

Dynamic
- Acceptance Testing
- Alpha Testing
- Assertion Checking
- Beta Testing
- Bottom-Up Testing
- Comparison Testing
- Compliance Testing
  - Authorization
  - Performance
  - Security
  - Standards
- Debugging
- Execution Testing
  - Monitoring
  - Profiling
  - Tracing
- Fault/Failure Insertion Testing
- Field Testing
- Functional (Black-Box) Testing
- Graphical Comparisons
- Interface Testing
  - Data
  - Model
  - User
- Object-Flow Testing
- Partition Testing
- Predictive Validation
- Product Testing
- Regression Testing
- Sensitivity Analysis
- Special Input Testing
  - Boundary Value
  - Equivalence Partitioning
  - Extreme Input
  - Invalid Input
  - Real-Time Input
  - Self-Driven Input
  - Stress
  - Trace-Driven Input
- Statistical Techniques
- Structural (White-Box) Testing
  - Branch
  - Condition
  - Data Flow
  - Loop
  - Path
  - Statement
- Submodel/Module Testing
- Symbolic Debugging
- Top-Down Testing
- Visualization/Animation

Formal
- Induction
- Inference
- Logical Deduction
- Inductive Assertions
- Lambda Calculus
- Predicate Calculus
- Predicate Transformation
- Proof of Correctness

Figure 4-1. A Taxonomy of Verification and Validation Techniques

4.1.1.1 Audit

An audit is undertaken to assess how adequately the application of M&S is conducted with respect to established plans, policies, procedures, standards, and guidelines. The audit also seeks to establish traceability within the simulation. When an error is identified, it should be traceable to its source via its audit trail. The process of documenting and retaining sufficient evidence about the substantiation of accuracy is called an *audit trail* (Perry, 1995).

Auditing is carried out periodically through a mixture of meetings, observations, and examinations (Hollocker, 1987). Audit is a staff function and serves as the "eyes and ears of management" (Perry, 1995, p. 26). In Verification, Validation, and Accreditation (VV&A), auditing is performed by the VV&A agent throughout the development life cycle for a new model or simulation or during modifications made to legacy models and simulations.

4.1.1.2 Desk Checking

Desk checking (also called *self-inspection*) is the process of intensely examining work to ensure its correctness, completeness, consistency, and clarity. It is considered to be the very first step in V&V and is particularly useful for the early stages of development. To be effective, desk checking should be conducted carefully and thoroughly, preferably by another person, because it is usually difficult to see one’s own errors (Adrion *et al.*, 1982). Syntax review, cross-reference examination, convention violation assessment, detailed comparison to specifications, code reading, control flowgraph analysis, and path sensitizing are all be conducted as part of desk checking (Beizer, 1990).

4.1.1.3 Face Validation

The project team members, potential users of the model, and people knowledgeable about the system of interest use their estimates and intuition to compare model and system behaviors subjectively under identical input conditions and judge whether the model and its results are reasonable (Hermann, 1967).

This technique is regularly cited in V&V efforts within DoD. It is one of the terms and techniques most commonly misused. Face validation is useful mostly as a preliminary approach to validation in the early stages of development. Except for a model that is mature and has an extensive, well-documented VV&A history, viable V&V efforts generally must use additional techniques.

4.1.1.4 Inspections

A team with four to six members inspects any M&S development phase such as M&S requirements definition, conceptual model design, or M&S detailed design. To inspect M&S design, for example, the team might consist of a moderator who manages the inspection team and provides leadership; a reader who narrates the M&S design and leads the team through the inspection process; a recorder who produces a written report of detected faults; a designer who represents the design developer; an implementer who translates the M&S design into an executable form; and a VV&A agent.

An inspection goes through five distinct phases: overview, preparation, inspection, rework, and follow-up (Schach, 1996). In Phase I, the designer summarizes the M&S design to be inspected. Characteristics such as problem definition, application requirements, and the specifics of software design are introduced and related documentation is distributed to all participants to study. In Phase II, the team members prepare individually for the inspection by examining the documents in detail. The success of the inspection rests heavily on the conscientiousness of the team members in their preparation. The moderator arranges the inspection meeting with an established agenda and chairs it in Phase III. The reader narrates the M&S design documentation and leads the team through the inspection process. The inspection team is aided during the fault-finding process by a checklist of queries. The objective is to find and document the faults, not to correct them. The recorder prepares a report of detected faults immediately after the meeting. In Phase IV, the designer resolves all faults and problems identified in the report. In the final phase, the moderator ensures that all faults and problems have been resolved satisfactorily. All changes must be examined carefully to ensure that no new errors have been introduced as a result of a fix.

Inspections have major differences from walkthroughs, described in Section 4.1.1.7. Briefly, a walkthrough is less formal, has fewer steps, and does not use a checklist to guide or a written report to document the team’s work. By comparison, an inspection is a five-step, formalized process. The inspection team uses the checklist approach for uncovering errors. The inspection process takes much longer than a walkthrough; however, the extra time is justified because an inspection is a powerful and cost-effective way of detecting faults early in the M&S development life cycle (Ackerman *et al.,* 1983; Beizer, 1990; Dobbins, 1987; Knight and Myers, 1993; Perry, 1995; Schach, 1996).

4.1.1.5 Reviews

The review is conducted similar to the inspection and walkthrough, except that the review team also involves managers. The review is intended to give management, such as the M&S proponent or the M&S application sponsor, evidence that the M&S development process is being carried out according to stated application objectives and to evaluate the model or simulation in light of development standards, guidelines, and specifications. As such, the review is a higher level technique than the inspection or walkthrough.

Each review team member examines the M&S documentation before the review. (Given the management positions of the team members, documentation needs to be less technical and more oversight-oriented than in an inspection. There also must be less material to examine if the V&V agent expects the team to prepare satisfactorily for the review.) The team then meets to evaluate the model or simulation relative to specifications and standards, recording defects and deficiencies. The review team may be given a set of indicators to measure such as (a) appropriateness of the problem definition and M&S requirements, (b) adequacy of all underlying assumptions, (c) adherence to standards, (d) modeling methodology used, (e) model representation quality, (f) model structure, (g) model consistency, (h) model completeness, and (i) documentation. (A checklist prepared by the V&V agent [not the developer!] is particularly useful in focusing management on the key points and in guiding the review.) The result of the review is a document portraying the events of the meeting, deficiencies identified, and review team recommendations. Appropriate action then may be taken to correct any deficiencies.

As opposed to inspections and walkthroughs, which concentrate on assessing correctness, reviews seek to ascertain that tolerable levels of quality are being attained. The review team is more concerned with model or simulation design deficiencies and deviations from stated model or simulation development policy than it is with the intricate line-by-line details of the implementation. This does not imply that the review team is not concerned with discovering technical flaws in the model or simulation, only that the review process is oriented toward the early stages of the M&S development life cycle (Hollocker, 1987; Perry, 1995; Sommerville, 1996; Whitner and Balci, 1989).

4.1.1.6 Turing Test

Turing test is based upon the expert knowledge of people about the system of interest. The experts are presented with two sets of output data, one obtained from the model and one from the system, under the same input conditions. Without identifying the data set, the experts are asked to differentiate between the two. If they succeed, they are asked to describe the differences. Their response provides valuable feedback for correcting model representation. If they cannot differentiate between the two, confidence in the model's validity is increased (Schruben, 1980; Turing, 1963; Van Horn, 1971).

4.1.1.7 Walkthroughs

A typical structured walkthrough team consists of a coordinator, often the V&V agent, who organizes, moderates, and follows up the walkthrough activities; a presenter, who is usually the model or simulation developer; a scribe who documents the events of the walkthrough meetings; a maintenance oracle who focuses on long-term implications; a standards bearer who assesses adherence to standards; the accreditation agent who reflects the needs and concerns of the accrediting authority; and other reviewers such as the model or simulation project manager and auditors. Except for the model or simulation developer, none of the team members should be involved directly in the development effort.

The main thrust of the walkthrough is to detect and document faults; it is *not* performance appraisal of the development team. This point must be made to everyone involved so that full cooperation is achieved in discovering errors.

The coordinator schedules the walkthrough meeting, distributes the walkthrough material to all participants well in advance to allow for careful preparation (again, critical to the success of the effort!), and chairs the meeting. During the meeting, the presenter narrates the walkthrough documents. (The V&V agent may wish to ascertain the level of preparation of the team members at the beginning of the meeting to ensure that materials have been read beforehand and that team members are not relying on the presenter's walkthrough of the material to obtain the information and insight needed for a meaningful discussion.) The coordinator encourages questions and discussion to uncover any faults (Adrion *et al.,* 1982; Deutsch, 1982; Myers, 1978, 1979; Yourdon, 1985).

The reader is encouraged to re-read Sections 4.1.1.4 and 4.1.1.5 on inspections and reviews to ensure a full understanding of the differences among these three techniques.

4.1.2 Static V&V Techniques

Static V&V techniques assess the accuracy of the static model design and source code. Static techniques do not require machine execution of the model, but mental execution can be used. The techniques are very popular and widely used, and many automated tools are available to assist in the V&V process. The simulation language compiler is itself a static V&V tool.

Static V&V techniques can reveal a variety of information about the structure of the model, the modeling techniques used, data and control flow within the model, and syntactical accuracy (Whitner and Balci, 1989).

4.1.2.1 Cause-Effect Graphing

Cause-effect graphing addresses the question of what causes what in the model representation. It first identifies causes and effects in the system being modeled and then examines their representation in the model specification. For example, in the simulation of a traffic intersection, the following causes and effects may be identified: (a) the change of a light to red immediately causes the vehicles in the traffic lane to stop, (b) an increase in the duration of a green light causes a decrease in the average waiting time of vehicles in the traffic lane, and (c) an increase in the arrival rate of vehicles causes an increase in the average number of vehicles at the intersection.

As many causes and effects as possible are listed, and the semantics are expressed in a cause-effect graph. The graph is annotated to describe special conditions or impossible situations. Once the cause-effect graph has been constructed, a decision table is created by tracing back through the graph to determine combinations of causes that result in each effect. The decision table then is converted into test cases with which the model is tested (Myers, 1979; Pressman, 1996; Whitner and Balci, 1989).

4.1.2.2 Control Analysis

Control analysis techniques consist of calling structure analysis, concurrent process analysis, control flow analysis, and state transition analysis.

Calling structure analysis is used to assess model accuracy by identifying who *calls* whom and who is *called* by whom. The *who* could be a procedure, subroutine, function, method, or a submodel within a model. For example, inaccuracies caused by message passing (e.g., sending a message to a nonexistent object) in an object-oriented model can be revealed by analyzing the specific messages that invoke an action and the actions that messages invoke (Miller *et al.,* 1995).

Concurrent process analysis is especially useful for parallel (Fujimoto, 1990, 1993; Page and Nance, 1994) and distributed simulations. If a simulation executes on a single computer with a single processor (CPU), it is referred to as a *serial (sequential) simulation*. If a single computer with multiple processors is used to execute the simulation model, then the simulation is said to be a *parallel simulation*. If multiple single-processor computers are used to execute the simulation model, then the simulation is said to be a *distributed simulation.*

Model accuracy is assessed by analyzing the overlap or simultaneous execution of actions executed in parallel or across distributed simulations. Such analysis can reveal synchronization and time management problems (Rattray, 1990).

Control flow analysis requires the graphing of the model, in which conditional branches and model junctions are represented by nodes and the model segments between such nodes are represented by links (Beizer, 1990). A node of the model graph usually represents a logical junction where the flow of control changes, whereas an edge represents the junction that assumes control. This technique examines sequences of control transfers and is useful for identifying incorrect or inefficient constructs within model representation.

State transition analysis identifies the finite number of states through which the model execution passes. A state transition diagram, which shows how the model transitions from one state to another, is created. Model accuracy is assessed by analyzing the conditions under which a state change occurs. This technique is especially effective for those M&S applications created under the activity scanning, three-phase, and process interaction conceptual frameworks (Balci, 1988).

4.1.2.3 Data Analysis

The data analysis category of V&V techniques consists of Data Dependency Analysis and Data Flow Analysis. These techniques are used to ensure that (a) proper operations are applied to data objects (e.g., data structures, event lists, linked lists), (b) the data used by the model are properly defined, and (c) the defined data are properly used (Perry, 1995).

Data dependency analysis determines which variables depend on other variables (Dunn, 1984). For parallel and distributed simulations, the data dependency knowledge is critical for assessing the accuracy of synchronization across multiple processors.

Data flow analysis assesses model accuracy with respect to the use of model variables. This assessment is classified according to the definition, referencing, and unreferencing of variables (Adrion *et al.,* 1982), i.e., when variable space is allocated, accessed, and deallocated. A data flowgraph is constructed to aid in the data flow analysis. The nodes of the graph represent statements and corresponding variables. The edges represent control flow.

Data flow analysis can be used to detect undefined or unreferenced variables (much as in static analysis) and, when aided by model instrumentation, can track minimum and maximum variable values, data dependencies, and data transformations during model execution. It is also useful in detecting inconsistencies in data structure declaration and improper linkages among submodels or federates (Allen and Cocke, 1976; Whitner and Balci, 1989).

4.1.2.4 Fault/Failure Analysis

Fault (incorrect model component) and failure (incorrect behavior of a model component) analysis uses model input-output transformation descriptions to identify how the model logically *might* fail. The model design specification is examined to determine if any failures logically could occur, in what context, and under what conditions. Such examinations often lead to identification of model defects (Miller *et al.,* 1995).

4.1.2.5 Interface Analysis

Interface analysis consists of model interface analysis and user interface analysis. These techniques are especially useful for verification and validation of interactive and distributed simulations.

Model interface analysis examines submodel-to-submodel interfaces within a model, or federate-to-federate interfaces within a federation, and determines if the interface structure and behavior are sufficiently accurate.

User interface analysis examines the user-model interface and determines if it is human engineered to prevent errors during the user's interactions with the model. It also assesses how accurately this interface is integrated into the overall model or simulation.

4.1.2.6 Semantic Analysis

Semantic analysis is conducted by the simulation programming language compiler and determines the modeler's intent as reflected by the code. The compiler describes the content of the source code so the modeler can verify that the original intent is reflected accurately.

The compiler generates a wealth of information to help the modeler determine if the true intent is translated accurately into the executable code: (a) s*ymbol tables,* which describe the elements or symbols that are manipulated in the model, function declarations, type and variable declarations, scoping relationships, interfaces, and dependencies; (b) c*ross-reference tables*, which describe called versus calling routines (where each data element is declared, referenced, and altered), duplicate data declarations (how often and where occurring), and unreferenced source code; (c) s*ubroutine interface tables,* which describe the actual interfaces of the caller and the called; (d) *maps,* which relate the generated runtime code to the original source code; and (e) *pretty printers* or s*ource code formatters,* which reformat the source listing on the basis of its syntax and semantics, clean pagination, highlighting of data elements, and marking of nested control structures (Whitner and Balci, 1989).

4.1.2.7 Structural Analysis

Structural analysis examines the model structure and determines if it adheres to structure principles. It is conducted by constructing a control flowgraph of the model structure and examining the graph for anomalies, such as multiple entry and exit points, excessive levels of nesting within a structure, and questionable practices such as the use of unconditional branches (i.e., GOTOs).

Yucesan and Jacobson (1992, 1996) apply the theory of computational complexity and show that the problem of verifying structural properties of M&S applications is difficult to solve. They illustrate that modeling issues such as accessibility of states, ordering of events, ambiguity of model specifications, and execution stalling are problems for which general design techniques do not produce efficient solutions.

4.1.2.8 Symbolic Evaluation

Symbolic evaluation assesses model accuracy by exercising the model using symbolic values rather than actual data values for input. It is performed by feeding symbolic inputs into the submodel or federate and producing expressions for the output that are derived from the transformation of the symbolic data along model execution paths. Consider, for example, the following function:

```
function jobArrivalTime(arrivalRate,currentClock,randomNumber)
      lag = -10
      Y = lag * currentClock
      Z = 3 * Y
      if Z < 0 then
            arrivalTime = currentClock – log(randomNumber) /
                  arrivalRate
      else
            arrivalTime = Z – log(randomNumber) / arrivalRate
      end if
      return arrivalTime
end jobArrivalTime
```

In symbolic execution, lag is substituted in Y resulting in Y = (`–10*currentClock`). Substituting again, Z is found to be equal to (`–30*currentClock`). Since `currentClock` is always zero or positive, an error is detected in that Z will never be greater than zero, and the “if-then-else” statement is unnecessary.

When unresolved conditional branches are encountered, a path is chosen to traverse. Once a path is selected, execution continues down the new path. At some point, the execution evaluation will return to the branch point and the previously unselected branch will be traversed. All paths eventually are taken.

The result of the execution can be represented graphically as a symbolic execution tree (Adrion *et al.,* 1982; King, 1976). The branches of the tree correspond to the paths of the model. Each node of the tree represents a decision point in the model and is labeled with the symbolic values of data at that juncture. The leaves of the tree are complete paths through the model and depict the symbolic output produced.

Symbolic evaluation assists in showing path correctness for all computations regardless of test data and is also a great source of documentation, but it has the following disadvantages: (a) the execution tree can explode in size and become too complex as the model grows; (b) loops cause difficulties although inductive reasoning and constraint analysis may help; (c) loops make thorough execution impossible because all paths must be traversed; and (d) complex data structures may have to be excluded because of difficulties in symbolically representing particular data elements within the structure (Dillon, 1990; King, 1976; Ramamoorthy *et al.*, 1976).

4.1.2.9 Syntax Analysis

Syntax analysis is carried on by the simulation programming language compiler to ensure that the mechanics of the language are applied correctly (Beizer, 1990).

4.1.2.10 Traceability Assessment

Traceability assessment is used to match, one to one, the elements of one form of the model to another. For example, the elements of the system as described in the requirements specification are matched one to one to the elements of the model or simulation design specification. Unmatched elements *may* reveal either unfulfilled requirements or unintended design functions (Miller *et al.*, 1995).

4.1.3 Dynamic V&V Techniques

Dynamic V&V techniques require model execution; they evaluate the model based on its execution behavior. Most dynamic V&V techniques require *model instrumentation*, the insertion of additional code (probes or stubs) into the executable model to collect information about model behavior during execution. Probe locations are determined manually or automatically based on static analysis of the model's structure. Automated instrumentation is accomplished by a preprocessor that analyzes the model's static structure (usually via graph-based analysis) and inserts probes at appropriate places.

Dynamic V&V techniques usually are applied in three steps. In Step 1, the executable model is instrumented. In Step 2, the instrumented model is executed; in Step 3, the model output is analyzed and dynamic model behavior is evaluated.

For example, consider the worldwide air traffic control and satellite communication object-oriented visual M&S application created by using the Visual Simulation Environment (Balci *et al.,* 1995) in Figure 4-2. The model can be instrumented in Step 1 to record the following information every time an aircraft enters into the coverage area of a satellite: (a) aircraft tail number; (b) time; (c) aircraft's longitude, latitude, and altitude; and (d) satellite's position and identification number. In Step 2, the model is executed and the information collected is written to an output file. In Step 3, the output file is examined to reveal discrepancies and inaccuracies in model representation.

4.1.3.1 Acceptance Testing

Acceptance testing is conducted by either the M&S application sponsor and the sponsor's VV&A agents or the developer's quality control group in the presence of the sponsor's representatives. The model is operationally tested with the actual hardware and data to determine whether all requirements specified in the legal contract are satisfied (Perry, 1995; Schach, 1996).

4.1.3.2 Alpha Testing

Alpha testing is the operational testing of the initial version of the complete model by the developer at an in-house site uninvolved with the model development (Beizer, 1990).

4.1.3.3 Assertion Checking

An assertion is a statement that should hold true as the simulation executes. Assertion checking is a verification technique that checks what is happening against what the modeler assumes is happening to guard against potential errors. The assertions are placed in various parts of the model to monitor execution. They can be inserted to hold true globally, for the whole model; regionally, for some submodels; locally, within a submodel; or at entry and exit of a submodel.

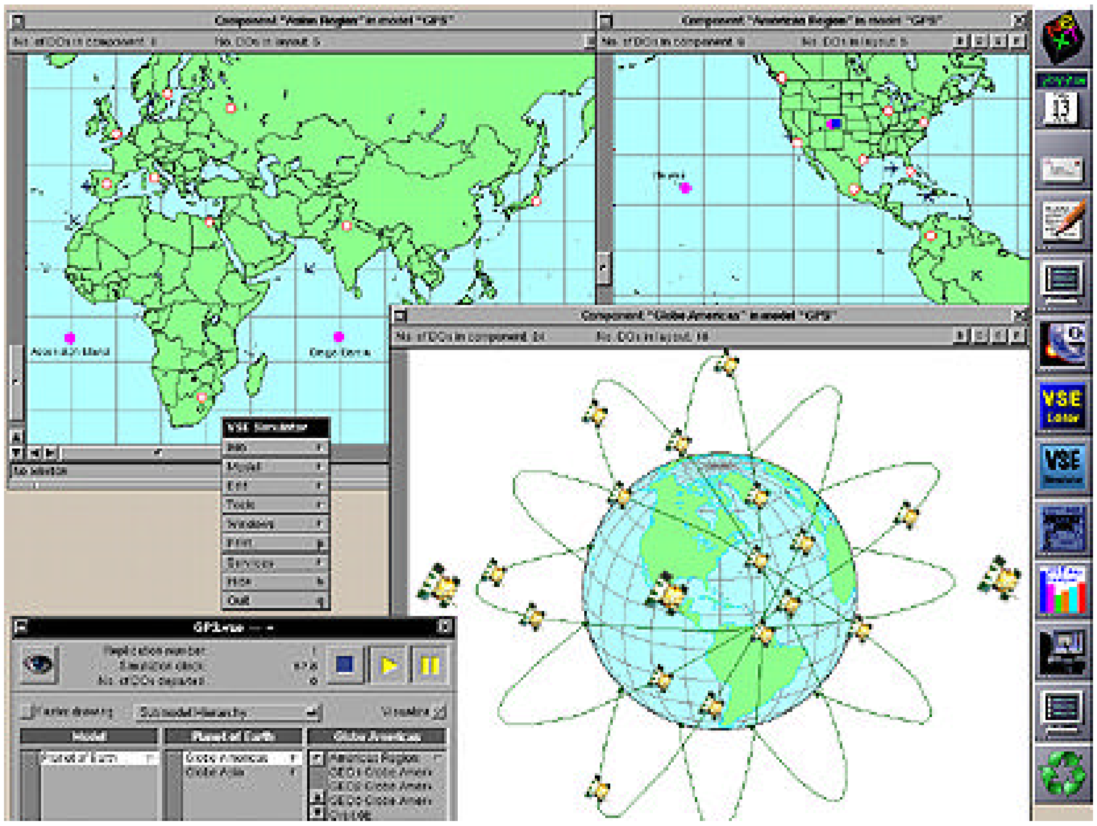

Figure 4-2. Visual Simulation of Global Air Traffic Control and Satellite Communication. (Reprinted from Balci *et al*., 1995.)

Consider, for example, the following pseudo-code (Whitner and Balci, 1989):

```
Base := Hours * PayRate;
Gross := Base * (1 + BonusRate);
```

In just these two simple statements, several assumptions are being made. It is assumed that `Hours`, `PayRate`, `Base`, `BonusRate`, and `Gross` are all non-negative. The following asserted code can be used to prevent execution errors caused by incorrect values entered by the user:

```
Assert Local (Hours > 0 and PayRate > 0 and BonusRate > 0);
Base := Hours * PayRate;
Gross := Base * (1 + BonusRate).
```

Assertion checking also prevents structural model inaccuracies. For example, the model in Figure 4-2 can contain assertions such as (a) a satellite communicates with the correct ground station, (b) an aircraft's tail number matches its type, and (c) an aircraft's flight path is consistent with the official airline guide.

Clearly, assertion checking serves two important needs: (a) it verifies that the model is functioning within its acceptable domain, and (b) the assertion statement documents the intentions of the modeler. Assertion checking, however, degrades model performance, forcing the modeler to choose between execution efficiency and accuracy. If the execution performance is critical, the assertions should be turned off but kept permanently in code to provide both documentation and means for maintenance testing (Adrion *et al.*, 1982).

4.1.3.4 Beta Testing

Beta testing refers to the developer's operational testing of the first-release version of the complete model at a beta user site under realistic field conditions (Miller *et al.*, 1995).

4.1.3.5 Bottom-Up Testing

Bottom-up testing is used with bottom-up model development. In bottom-up development, model construction starts with the simulation's routines at the base level, i.e., the ones that cannot be decomposed further, and culminates with the submodels at the highest level. As each routine is completed, it is tested thoroughly. When routines with the same parent, or submodel, have been developed and tested, the routines are integrated and their integration is tested. This process is repeated until all submodels and the model as a whole have been integrated and tested. The integration of completed submodels need not wait for all submodels at the same level to be completed. Submodel integration and testing can be, and often is, performed incrementally (Sommerville, 1996).

Some of the advantages of bottom-up testing are (a) it encourages extensive testing at the routine and submodel levels; (b) because most well-structured models consist of a hierarchy of submodels, much may be gained by bottom-up testing; (c) the smaller the submodels and the more cohesion within the model, the easier and more complete its testing will be; and (d) it is particularly attractive for testing distributed models and simulations.

Major disadvantages of bottom-up testing include (a) individual submodel testing requires drivers, more commonly called test harnesses, which simulate the calling of the submodel and passing test data necessary to execute the submodel; (b) developing harnesses for every submodel can be quite complex and difficult; (c) the harnesses may themselves contain errors; and (d) bottom-up testing faces the same cost and complexity problems as does top-down testing (see Section 4.1.3.26).

4.1.3.6 Comparison Testing

Comparison testing (also known as back-to-back testing) may be used when more than one version of a model or simulation representing the same system is available for testing (Pressman, 1996; Sommerville, 1996). For example, different simulations may have been developed by the different Services to simulate the same military combat aircraft. (The development of the High-Level Architecture (HLA), however, is intended to reduce greatly such redundant model development in favor of fewer simulations at less cost to DoD.) All simulations built to represent exactly the same system are run with the same input data and the model outputs are compared. Differences in the outputs reveal problems with model accuracy. The major disadvantage to this technique is the lack of information that generally exists about the validity of the other models. If two models both were written with a specific, unnoticed error in the code, the results might agree but would still be invalid.

4.1.3.7 Compliance Testing

Compliance testing compares the simulation to required security and performance standards. These techniques are particularly useful for testing federations of distributed and interactive models and simulations. Compliance testing methods for HLA compliance have been developed and are available from DMSO.

Authorization testing tests how accurately different levels of security access authorization are implemented in the simulation and how properly they comply with established rules and regulations. The test can be conducted by attempting to execute a classified model within a federation or by using classified input data to run a simulation without proper authorization (Perry, 1995).

Performance testing simply tests whether all performance characteristics are measured and evaluated with sufficient accuracy and if all established performance requirements are satisfied (Perry, 1995).

Security testing tests whether all security procedures are implemented correctly and properly. For example, penetrating the simulation while it is running and breaking into classified components such as secure databases can be attempted. Security testing evaluates the adequacy of protective procedures and countermeasures (Perry, 1995).

Standards testing substantiates that the M&S application is developed with respect to the required standards, procedures, and guidelines.

4.1.3.8 Debugging

Debugging is an iterative process that uncovers errors or misconceptions that cause the model's failure and defines and carries out the model changes that correct the errors. This iterative process consists of four steps. In Step 1, the model is tested, revealing the existence of errors (bugs). Given the errors detected, the cause of each error is determined in Step 2. In Step 3, the model changes necessary to correct the detected errors are identified and are carried out in Step 4. Step 1 is re-executed immediately after Step 4 to ensure successful modification, because a change correcting an error may create another one. This iterative process continues until no errors are identified in Step 1 after sufficient testing (Dunn, 1987).

4.1.3.9 Execution Testing

Execution testing consists of monitoring, profiling, and tracing techniques. These techniques collect and analyze execution behavior data to reveal model representation errors.

Execution monitoring reveals errors by examining low-level information about activities and events that take place during model execution. It requires the instrumentation of a model or simulation to gather data to provide activity- or event-oriented information about the model's dynamic behavior. For example, the model in Figure 4-2 can be instrumented to monitor the arrivals and departures of aircraft within a particular city, and the results can be compared with the official airline guide to judge model validity. The model also can be instrumented to provide other low-level information such as the number of late arrivals, the average passenger waiting time at the airport, and the average flight time between two locations.

Execution profiling reveals errors by examining high-level information (profiles) about activities and events that take place during model execution. It requires the instrumentation of an executable model to gather data to present profiles about the model's dynamic behavior. For example, the model in Figure 4-2 can be instrumented to produce histograms of aircraft departure times, arrival times, and passenger check-out times at an airport.

Execution tracing reveals errors by reviewing the line-by-line execution of a simulation. It requires the instrumentation of an executable model to trace the model's line-by-line dynamic behavior. The model in Figure 4-2 can be instrumented to record all aircraft arrival times at a particular airport. Then, the trace data can be compared with the official airline guide to assess model validity.

The major disadvantage of the tracing technique is that execution of the instrumented model may produce a large volume of trace data too complex to analyze. To overcome this problem, the trace data can be stored in a data base and the modeler can analyze it using a query language (Fairley, 1975, 1976).

4.1.3.10 Fault/Failure Insertion Testing

This technique inserts a fault (incorrect model component) or a failure (incorrect behavior of a model component) into the model and observes whether the model produces the invalid behavior as expected. Unexplained behavior may reveal errors in model representation.

4.1.3.11 Field Testing

Field testing places the model in an operational situation and collects as much information as possible for model validation. It is especially useful for validating models of military combat systems. Field testing conducted as part of the test and evaluation process is particularly important within DoD system acquisition. It is a major element of VV&A conducted during the development of new weapons systems and platforms. Although it is usually difficult, expensive, and sometimes impossible to devise meaningful field tests for complex systems, their use wherever possible helps both the project team and decision makers develop confidence in the model (Shannon, 1975; Van Horn, 1971). The greatest disadvantage of field testing is the lack of adequate test resources to produce statistically significant results. Often, simulation runs augment live test data in the development and decision processes.

4.1.3.12 Functional Testing

Functional testing (also called *black-box testing*) assesses the accuracy of model input-output transformation. It is applied by feeding inputs (test data) to the model and evaluating the accuracy of the corresponding outputs.

It is virtually impossible to test all input-output transformation paths for a reasonably large and complex simulation, because the number of those paths could be in the millions. Therefore, the objective of functional testing is to increase confidence in model input-output transformation accuracy as much as possible rather than to claim absolute correctness.

The generation of test data is a crucially important but very difficult task. The law of large numbers does not apply here. Successfully testing the model under 1,000 input values (test data) does not imply high confidence in model input-output transformation accuracy just because of the large number. Instead, the number 1,000 should be compared with the number of allowable input values to determine the percentage of the model input domain that is covered in testing. The more the model input domain is covered in testing, the more confidence is gained in the accuracy of the model input-output transformation (Howden, 1980; Myers, 1979).

4.1.3.13 Graphical Comparison

Graphical comparison is a subjective, inelegant, and heuristic, yet quite practical approach, especially useful as a preliminary step to model V&V. The graphs of values of model variables over time are compared with the graphs of values of system variables to investigate characteristics such as similarities in periodicities, skewness, number, and location of inflection points; logarithmic rise and linearity; phase shift; trend lines; and exponential growth constants (Cohen and Cyert, 1961; Forrester, 1961; Miller, 1975; Wright, 1972).

4.1.3.14 Interface Testing

Interface testing (also known as *integration testing*) tests the data, model, and user interfaces. Interface testing is more rigorous than the interface analysis discussed in Section 4.1.2.5.

Data interface testing assesses the accuracy of data entered into the model or derived from the model during execution. All data interfaces are examined to substantiate that all aspects of data input and output are correct. This form of testing is particularly important for those simulations in which the inputs are read from a data base or the results are stored in a data base for later analysis. The model's interface to the data base is examined to ensure correct importing and exporting of data (Miller *et al.*, 1995). Data interface testing is key to the relationship between the VV&A effort and the corresponding Verification, Validation, and Certification (VV&C) of data effort.

Model interface testing detects model representation errors created as a result of submodel-to-submodel or federate-to-federate interface errors or invalid assumptions about the interfaces. It is essential that each submodel within a model or model (federate) within a federation is tested individually and found to be sufficiently accurate before model interface testing begins. (Recall Principle 6 from Chapter 2!)

This form of testing deals with how well the submodels (or federates) are integrated with each other and is particularly useful for object-oriented and distributed simulations. Under the object-oriented paradigm, objects (a) are created with public and private interfaces, (b) interface with other objects through message passing, (c) are reused with their interfaces, and (d) inherit the interfaces and services of other objects.

Model interface testing assesses the accuracy of four types of interfaces, as identified by Sommerville (1996):

1. Parameter interfaces that pass data or function references from one object to another
2. Shared memory interfaces that enable objects to share a block of memory in which data are placed by one object and from which they are retrieved by other objects
3. Procedural interfaces that implement the concept of encapsulation under the object-oriented paradigm—an object provides a set of services (procedures) that can be used by other objects and hides (encapsulates) the way a service is provided from the outside world

4. Message-passing interfaces that enable an object to request the service of another object through message passing

Sommerville (1996) classifies interface errors into three categories:

1. Interface misuse occurs when an object calls another and incorrectly uses its interface. For objects with parameter interfaces, a parameter may be of the wrong type or may be passed in the wrong order, or the wrong number of parameters may be passed.
2. Interface misunderstanding occurs when object A calls object B without satisfying the underlying assumptions of object B's interface. For example, object A calls a binary search routine by passing an unordered list to be searched, when in fact the binary algorithm assumes that the list is already sorted.
3. Timing errors occur in real-time, parallel, and distributed simulations that use a shared memory or a message-passing interface.

User interface testing detects model representation errors created as a result of user-model interface errors or invalid assumptions about this interface. This form of testing is particularly important for testing human-in-the-loop and interactive simulations.

User interface testing assesses the interactions between the user and the model. The user interface is examined from low-level ergonomic aspects to instrumentation and controls and from human factors to global considerations of usability and appropriateness to identify potential errors (Miller *et al.,* 1995; Pressman, 1996; Schach, 1996).

4.1.3.15 Object-Flow Testing

Object-flow testing is similar to *transaction-flow testing* (Beizer, 1990) and *thread testing* (Sommerville, 1996). It assesses model accuracy by exploring the life cycle of an object during model execution. For example, a dynamic object (aircraft) can be marked for testing in the visual simulation environment for the model shown in Figure 4-2. Every time the dynamic object enters into a subroutine, the visualization of that subroutine is displayed. Every time the dynamic object interacts with another object within the subroutine, the interaction is highlighted. Examination of the way a dynamic object flows through the activities and processes and interacts with its environment during its lifetime in model execution is extremely useful for identifying errors in model behavior.

4.1.3.16 Partition Testing

Partition testing examines the model with the test data generated by analyzing the model's functional representations or partitions. It is accomplished by (a) decomposing both the model specification and its implementation into functional representations (partitions), (b) comparing the elements and prescribed functionality of each partition specification with the elements and actual functionality of the corresponding partition as it has been implemented in code, (c) deriving test data to test the functional behavior of each partition extensively, and (d) testing the model with the generated test data.

The model is decomposed into functional representations (partitions) through the use of symbolic evaluation techniques that maintain algebraic expressions of model elements and show model execution paths. These functional representations are the model computations. Two computations are equivalent if they are defined for the same subset of the input domain that causes a set of model paths to be executed and if the result of the computations is the same for each element within the subset of the input domain (Howden, 1976). Standard proof techniques show equivalence over a domain. When equivalence cannot be shown, partition testing is performed to locate errors or, as Richardson and Clarke (1985, p. 1488) state, to "increase confidence in the equality of the computations due to the lack of error manifestation." By involving both the model's specification and its implementation, partition testing can provide more comprehensive test data coverage than other test data generation techniques.

4.1.3.17 Predictive Validation

Predictive validation requires past input and output data from the system being modeled. The model is driven by past system input data and its forecasts are compared with the corresponding past system output data to test the predictive ability of the model (Emshoff and Sisson, 1970). Test data from test and evaluation uses of M&S are one example of how this technique is often used. Predictive validation also can evolve into the Model-Test-Model methodology, which uses the test data to make subsequent improvements to the model.

4.1.3.18 Product Testing

Product testing is conducted by the model or simulation developer after all submodels are successfully integrated (as demonstrated by the interface testing) and before the acceptance testing is performed by the model or simulation application sponsor or proponent. No contractor wants the product (model) to fail the acceptance test. Product testing serves to prepare for the acceptance testing. As such, the developer’s quality control group must test the product and make sure that all requirements specified in the legal contract are satisfied before delivering the model to the model or simulation application sponsor (Schach, 1996).

As dictated by Principle 6 in Chapter 2, successfully testing each submodel or federate does not imply overall model or federation credibility. Interface testing and product testing are two techniques that must be performed to substantiate overall model credibility.

4.1.3.19 Regression Testing

Regression testing investigates the relationships between variables. In particular, it ensures that correcting errors and making changes in the model do not create other errors and adverse side effects. Usually the modified model is retested with the test data sets used previously. Successful regression testing requires the retention and management of old test data sets throughout the model development life cycle.

4.1.3.20 Sensitivity Analysis

Sensitivity analysis is performed by systematically changing the values of model input variables and parameters over some range of interest and observing the effect upon model behavior (Shannon, 1975). Unexpected effects may reveal invalidity. The input values also can be changed to induce errors to determine the sensitivity of model behavior to such errors. Sensitivity analysis can identify those input variables and parameters to which model behavior is very sensitive. Model validity then can be enhanced by ensuring that those values are specified with sufficient accuracy (Hermann, 1967; Miller, 1974a,b; Van Horn, 1971).

4.1.3.21 Special Input Testing

Special input testing consists of eight types of tests: boundary value, equivalence partitioning, extreme input, invalid input, real-time input, self-driven input, stress, and trace-driven input techniques. These techniques assess model accuracy by subjecting the model to a variety of inputs.

Boundary value testing examines the model's accuracy by using test cases on the boundaries of input equivalence classes. A model's input domain usually can be divided into classes of input data (known as equivalence classes) that cause the model to function the same way. For example, a traffic intersection model might specify the probability of left turn in a three-way turning lane as 0.2, the probability of right turn as 0.35, and the probability of traveling straight as 0.45. This probabilistic branching can be implemented by using a uniform random-number generator that produces numbers in the range $0 \leq rn \leq 1$. Thus, three equivalence classes are identified: $0 \leq rn \leq 0.2$, $0.2 < rn \leq 0.55$, and $0.55 < rn \leq 1$. Each test case from within a given equivalence class has the same effect on the model behavior, i.e., produces the same direction of turn.

In boundary analysis, test cases are generated just within, on top of, and outside of the equivalence classes (Myers, 1979). In the example above, the following test cases are selected for the left turn: 0.0, ±0.000001, 0.199999, 0.2, and 0.200001. In addition to generating test data on the basis of input equivalence classes, it also is useful to generate test data that will cause the model to produce values on the boundaries of *output* equivalence classes (Myers, 1979). The underlying rationale for this technique as a whole is that the most error-prone test cases lie along the boundaries (Ould and Unwin, 1986). Notice that invalid test cases used in the example will cause the model execution to fail; however, this failure should be as expected and meaningfully documented.

Equivalence partitioning testing partitions the model input domain into equivalence classes in such a manner that a test of a representative value from a class is assumed to be a test of all values in that class (Miller *et al.,* 1995; Perry, 1995; Pressman, 1996; Sommerville, 1996).

Extreme input testing is conducted by running the model or simulation with only minimum values, maximum values, or an arbitrary mixture of minimum and maximum values for the model input variables. For example, this technique allows the model user to test a proposed weapon system against extreme conditions that may not be obtainable in actual system testing.

Invalid input testing is performed by running the model or simulation under incorrect input data to determine whether the model behaves as expected. Unexplained behavior may reveal errors in model representations.

Real-time input testing is particularly important for assessing the accuracy of simulations built to represent embedded real-time systems. For example, different design strategies of a real-time software system built to control the operations of a manufacturing system can be studied using M&S. The model that represents the software design can be tested by running it with real-time input data that can be collected from the existing manufacturing system. Using real-time input data collected from a real system is particularly important to capture the timing relationships and correlations between input data points.

Self-driven input testing is conducted by running the model or simulation under input data randomly sampled from probabilistic models representing random phenomena in a real or future system. A probability distribution (e.g., exponential, gamma, weibull) can be fit to collected data, or triangular and beta probability distributions can be used in the absence of data, to model random input conditions (Banks *et al.*, 1996; Law and Kelton, 1991). Then, using random variate generation techniques, random values can be sampled from the probabilistic models to test the model validity under a set of observed or speculated random input conditions.

Stress testing tests the model's validity under extreme workload conditions. This is usually accomplished by increasing the congestion in the model. For example, the model in Figure 4-2 can be stress tested by increasing the number of flights between two locations to an extremely high value. Such an increase in workload may create unexpected high congestion in the model. Under stress testing, the model may exhibit invalid behavior; however, such behavior should be as expected and meaningfully documented (Dunn, 1987; Myers, 1979).

Trace-driven input testing is conducted by running the model or simulation under input trace data collected from a real system. For example, a system can be instrumented with monitors that collect data by tracing all system events. The raw trace data then are refined to produce the real input data for testing the model or simulation.

4.1.3.22 Statistical Techniques

Much research has been conducted in applying statistical techniques to model validation. Table 4-1 presents the statistical techniques proposed for model validation and lists related references.

Table 4-1. Statistical Techniques Proposed for Validation

Technique	*References*
Analysis of Variance	Naylor and Finger, 1967
Confidence Intervals/Regions	Balci and Sargent, 1984; Law and Kelton, 1991; Shannon, 1975
Factor Analysis	Cohen and Cyert, 1961
Hotelling's T^2 Tests	Balci and Sargent, 1981, 1982a, 1982b, 1983; Shannon, 1975
Multivariate Analysis of Variance —Standard MANOVA —Permutation Methods —Nonparametric Ranking Methods	Garratt, 1974
Nonparametric Goodness-of-Fit Tests —Kolmogorov-Smirnov Test —Cramer-Von Mises Test —Chi-square Test	Gafarian and Walsh, 1969; Naylor and Finger, 1967
Nonparametric Tests of Means —Mann-Whitney-Wilcoxon Test —Analysis of Paired Observations	Shannon, 1975
Regression Analysis	Aigner, 1972; Cohen and Cyert, 1961; Howrey and Kelejian, 1969
Theil's Inequality Coefficient	Kheir and Holmes, 1978; Rowland and Holmes, 1978; Theil, 1961
Time Series Analysis —Spectral Analysis —Correlation Analysis —Error Analysis	 Fishman and Kiviat, 1967; Gallant et al., 1974; Howrey and Kelejian, 1969; Hunt, 1970; Van Horn, 1971; Watts, 1969 Watts, 1969 Damborg and Fuller, 1976; Tytula, 1978
t-Test	Shannon, 1975; Teorey, 1975

The statistical techniques listed in Table 4-1 require the system being modeled to be completely observable, i.e., that all data required for model validation can be collected from the system. The model is validated by using the statistical techniques to compare the model output data with the corresponding system output data after the model is run with the same input data as the real system. Model and system outputs are compared using multivariate statistical techniques to capture the correlation among the output variables. A recommended validation procedure based on the use of simultaneous confidence intervals follows.

Example 4-1. A Validation Procedure Using Simultaneous Confidence Intervals.

The behavioral accuracy (validity) of a simulation with multiple outputs can be expressed in terms of the differences between the corresponding model and system output variables when the model is run with the same input data and operational conditions that drive the real system. The range of accuracy of the j*th model output variable can be represented by the* j*th confidence interval (c.i.) for the differences between the means of the* j*th model and system output variables. The simultaneous confidence intervals (s.c.i.) formed by these confidence intervals are called the model range of accuracy (m.r.a.) (Balci and Sargent, 1984).*

Assume that there are k *output variables from the model and* k *output variables from the system as shown in Figure 4.3. Let* $\left(\underline{\mu}^{m}\right)' = \left[\mu_1^m, \mu_2^m, ..., \mu_k^m\right]$ *and* $\left(\underline{\mu}^{s}\right)' = \left[\mu_1^s, \mu_2^s, \ldots, \mu_k^s\right]$ *be the* k *dimensional vectors of the population means of the model and system output variables, respectively. Basically, there are three approaches for constructing the s.c.i to express the m.r.a. for the mean behavior.*